LAP User's Manual

by

Dick Lang

Technical Report 3356

December 1979

Computer Science

California Institute of Technology

Pasadena, California 91125

Silicon Structures Project

sponsored by

Burroughs Corporation, Digital Equipment Corporation,

Hewlett-Packard Company, Honeywell Incorporated,

International Business Machines Corporation,

Intel Corporation, Xerox Corporation,

and the National Science Foundation

Introduction

LAP is a set of Simula procedures and objects declared externally for use in generating CIF 2.0. The user can use these procedures and any other Simula statments to programmatically generate chip geometry. Simula is syntactically an Algol-like language. The details and capabilities of Simula can be found in the Simula Langauge Handbook (Birtwistle, et al) and Simula Begin, (Birtwistle, et al). As with Algol, all variables must be declared, block structuring is available (BEGIN ... END;) and the various control structures may be used (FOR loops, IF THEN, WHILE DO). Users may, of course, declare their own procedures. In fact, users may do any damn thing they want.

To utilize LAP, the users should make use of the LAP.FORM file. This file contains all the external declarations necessary to use LAP. LAP statements and programs are inserted at the place indicated in this template file.

Writing LAP

For the most part LAP constructs correspond one-to-one with CIF 2.0 commands (See Mead and Conway Chapter 4.5). Not all of the CIF 2.0 geometric objects are available in LAP. All dimensions LAP are in lambda.

LAP lets you define symbols. A good symbol structure is essential to making a good project. In general, you should define a symbol for each recognizably different structure. Thus if you have a project which has two super-buffers in it, you will want a super-buffer symbol. You can then call on that symbol and place it at different locations in your layout. The trick is to recognize what should be a symbol and to define your symbols carefully.

To define a symbol in LAP do this:

```
DEFINE("name-of-symbol");
! put the specification of the symbol here;
ENDDEF;
```

The specification cannot include additional DEFINE statements but may include calls to previously defined symbols (forward references are not allowed) and geometric commands. Upper case and lower case text, as used in symbol names, are treated the same.

Calls to symbols may be done is several ways depending on the transformation that you want applied to the symbol.

The basic call with only translation is as follows:

```
DRAW("name-of-symbol",x,y);
```

Where x and y are reals indicating where the symbol origin is to be translated to. If you want translation and rotation:

```
DRAWROT("name-of-symbol",x,y,x-direction,y-direction);
```

Where x-direction and y-direction represent a vector which points in the direction of the new x-axis. A rotation vector of (1,0) does nothing since it points in the same direction as the x-axis. If you want translation and mirroring then:

```
DRAWMX("name-of-symbol",x,y);     ! mirror in x ;
DRAWMY("name-of-symbol",x,y);     ! mirror in y ;
```

Mirroring a symbol causes all coordinates of the indicated type to be negated. That is, if DRAWMX is used all x-coordinates are negated prior to translation.

Geometric Commands

There are three basic commands in this group: wires, boxes and layers. The layer command specifies which of several layers succeeding wires and boxes will be places on. A layer command must precede any wires or boxes in a given symbol definition. Layer commands do not affect symbol calls in a given symbol definition. The layer command is shown below:

```
LAYER(layer-name);
```

The layer-names are defined as follows:

NMOS layer	acceptable names
metal	metal, blue
polysilicon	poly, red
diffusion	diffusion, green
contact cuts	cuts, black
ion implantaion	implant, yellow
over glass	glass

To make a box on the current layer, the box command is used.

```
BOX(x1,y1,x2,y2);
```

Where x1,y1 is the point representing the lower left corner of the box and x2,y2 is a point representing the upper right corner.

The wire command is used to generate CIF wires. It begins as follows:

```
WIRE(width,x1,y1)....
```

Where width is the width of the wire in lambda and x1,y1 is the initial starting point of the wire. Immediately following the close-parenthesis the remaining points of the wire's path are specified using the following extensions:

```
WIRE(width,x1,y1).x(x2).y(y2).xy(x3,y3);
```

Each extension (a dot followed by a procedure) generates a new point in the wires path. Thus x(x2) generates the second point x2,y1 y(y2) generates x2,y2 and xy(x3,y3) generates x3,y3. Also avaibable are the extensions:

```
.dx(x)
.dy(y)
.dxy(x,y);
```

Each of these adds their parameter to the last value of the proper coordinate allowing the points of the wire to be specified relative to the starting point of the wire.

The extensions may be used in any order with any parameters (parameters may be expressions or literals). More than 8 extensions will cause an error in the compilation of the Simula. If a wire with more than 8 points is needed then a new wire must be started where the previous one ended.

Two additional extensions are available for wires:

```
.w(width)
.z(layer)
```

The first allows the user to change the width of the wire. This is most useful when changing layers to convert from the 3 lambda minimum width of metal wires to the 2 lambda minimum width of poly and diffusion wires. The "z" extension allows the wire to move between the poly, diffusion and metal layers. The paramenter must specify one of these three layers. Lap will build a contact to the desired layer at the most previous point in the wire's path. Additional extensions to the wire will draw geometry on the new layer. The current layer is changed by the "z" extension and will apply to any subsequent box or wire commands until a new layer command is encountered.

Predefined Procedures and Symbols

--- --- --- ---

Lap has some built-in facilities for generating contacts and pads. There are 6 procedures which may be invoked anywhere in a symbol definition to make contact between layers. These procedures shall be the ONLY methods used to generate contacts. Using them prevents one from forgeting one or more components of the contact and insures that their sizes are correct.

rb(x,y)	red to blue contact centered at x,y
gb(x,y)	green to blue contact
be(x,y)	green to red butting contact, red facing east
bs(x,y)	green to red butting contact, red facing south
bw(x,y)	green to red butting contact, red facing west
bn(x,y)	green to red butting contact, red facing north

The origin of the butting contact commands is on the centerline of the long axis of the symbol one lambda in from the green end.

A symbol is predefined for generating pads, it is the ONLY symbol that should be used to generate pads. It is invoked by:

```
DRAW("pad",x,y);
```

A metal pad will then be drawn centered at x,y 42 lambda on a side with a hole above it in the over glass.

Many times it is necessary to draw text on the chip masks. Lap has a procedure for doing this:

```
ALPHA("alpha-numeric-characters",x,y,scale);
```

The quoted string is drawn beginning a x,y and extends to the right. The scale factor multiplies all the dimensions of the generated geometry to allow the characters drawn to be enlarged. With a scale factor of 1, the size of the characters will be 9 lambda tall by 7 wide. The point x,y will be the lower left corner of the bounding box of the first character. The ALPHA procedure acts like a list of wire commands and may be used inside any symbol definition.

There is also a mechanism for drawing pullup transistors (depletion loads):

```
PULLUP(PATH(x1,y1)...path extensions...)...path extensions;
```

This draws a pullup transistor complete with implant and butting contact beginning a x1,y1. At this point the butting contact is drawn and x,y,xy,dx,dy and dxy extensions are used to specify the path of minimum width diffusion wire. Poly and implant are properly drawn about the channel region specified by the diffusion wire path. The transistor channel will stop at the last point in the path. The diffusion wire (drain) is extended

away from the transistor by the second set of extensions. This portion acts the same as any other wire command.

CIF Cell Libraries

One LAP program may reference CIF files generated by other LAP programs using the READCIF command. The READCIF command may be executed at any time, it reads the indicated file and extracts from it the names, numbers and bounding boxes of all the CIF symbols within the file. These symbols are then accessable to the LAP program which executed the command. No CIF is generated for these symbols, as it is assumed that the user will at some point concatenate the various CIF files together prior to using the CIF. The file read by the command must not have symbol number or symbol name conflicts with existing symbols already defined by the user. To insure this, the easiest method is to begin the LAP program with all necessary READCIF commands followed by the usual LAP commands which may then utilize the symbols seen on the files. The compilation of CIF from LAP will be faster and permit larger designs to be run if the READCIF command is used. The format of the READCIF command is as follows:

```
READCIF("filename");
```

The single parameter is the name of a file enclosed in quotes. If the filename is entered with no extension, an extension of .CIF is assumed.

The use of several indepenent CIF library files can cause symbol number conflicts if steps are not taken to coerce the assignment of symbol numbers by LAP. A global variable is available to the user. Its name is SEQ. Setting SEQ determines the next number assigned as a symbol number in the execution of a DEFINE. The variable SEQ is incremented and then used by DEFINE such that SEQ:=100 will cause the next symbol defined to be numbered 101.

At some point, the set of CIF library files will be concatenated into one file by the user. To avoid having to edit out the CIF "E" command at the end of CIF files, LAP can be prevented from writing the "E" command. The statement NOEND:=TRUE; will cause LAP to not append an E to the end of the CIF file. The absence of the statement will cause the E to be written.

Simula Notes

All Simula statments must be terminated with a semi-colon. Comments in Simula are begun with an exclamation point (!) and ended with a semi-colon (;).

Documentation for CLASS PLA

CLASS PLA is a built-in LAP program which draws synchronous Programmable Logic Arrays. There are 6 input parameters to the PLA:

The number of inputs to the PLA, including feedback terms,
The number of minterms (connections between the AND and OR planes),
The number of outputs from the PLA, again including the feedbacks,
The number of feedback terms (outputs which directly connect to inputs of the same PLA), and
The name of a file which contains the coding information for the PLA.
The integer value for the PLAguts switch which controls how much of the PLA gets drawn. (Ex.8=whole PLA incl. clock, 2=just bristle connections, etc)

There may be any number of inputs to the PLA, but there must be at least as many inputs as there are feedback terms. There may be any even number of minterms in the PLA. If your PLA only needs an odd number of minterms, ask for the next largest even number instead (if you need 5 minterms, ask for 6 instead). The number of outputs for the PLA must also be an even number, and you should have at least as many outputs as feedback terms. The file name should not include directory information or extensions. The file should either be in your directory or on your search path. The program assumes an extension of ".COD" for all PLA code files.

The feedback terms connect the last input to the first output, the second-to-last input to the second output, etc. The remaining input terms require diffusion wires providing the input signals. There is a procedure named PLAINPUT(n) which returns the location (a POINT) of the nth PLA input. For example:

```
ref(pla) pla1;
pla1:=new pla(3,14,6,2,"test",8);
layer(green);
wire(2,....).....xy(pla1.plainput(1));
```

generates a new pla with 3 inputs, 14 minterms, 6 outputs, 2 feedbacks, and which reads the file "test.cod" to find the code. The last line draws a diffusion wire which connects to the first PLA input.

There is a second procedure which returns the location of the nth PLA output. (remember, the first outputs are used as feedbacks, so the last outputs are the actual PLA outputs). The outputs are available in either diffusion or metal, and if you want to connect to a feedback term, you must use metal (unless it is the last feedback term). PLAOUTPUT(n) returns the location of output number n.

The PLA also requires two phases of clock. Phase 1, which gates the inputs, should connect in polysilicon to location PHI1, while Phase 2, which gates the outputs, should connect to PHI2. These are both procedures which return the location of the respective signal.

Vdd connects to the left edge of the PLA, and Ground connects to the right edge. The procedure MBB returns a rectangle containing the lower left and upper right points of the bounding box.

The format of the code file is as follows. There is a single line for each minterm of the PLA. On each line, the coding for the AND plane and coding for the OR plane is given. The coding for the AND plane uses characters from the set (X,1,0), one for each input to the PLA. Following the AND plane code is one blank space, followed by the OR plane code. The OR plane uses characters from the set (1,0), one for each output. For our example PLA, the line:

```
    0X1 011100
```

will drive outputs 2, 3, and 4 high whenever input 1 is low and input 3 is high.

The PLAguts switch setting works as follows :

```
IF PLAguts > 7 then draws the whole PLA
   ELSE IF PLAguts > 5 then draw whole PLA but without clock
      ELSE IF PLAguts > 3 then draw bristles and
            inner transistors(.cod).
         ELSE IF PLAguts > 1 then draw bristles only (incl clock)
            ELSE IF PLAguts > -1 then draw boundary
                  wires only(has bug)
               ELSE draw nothing but do x,y calculations ;
```

As an example, we will build a PLA with 6 inputs, 8 minterms, 4 outputs, and 2 feedbacks. The file "TEST.COD" will contain the code for the PLA, and looks like this:

```
000000 0010
01XX10 1000
0X1X01 0110
1XXXXX 1111
000110 0001
010101 0100
0XXX11 0101
```

```
X0X1X0 0111
```

In the declarations portion of our LAP program, we declare "PLA1" to be a reference to a PLA:

```
REF (pla) PLA1;
```

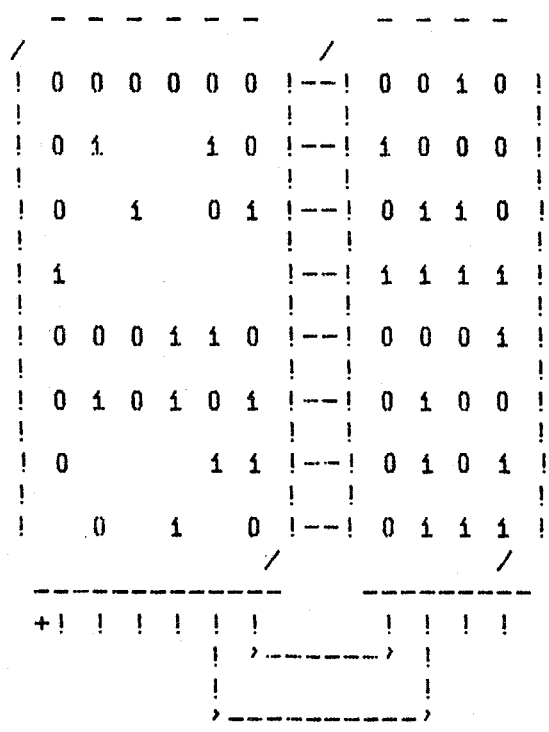
and in the execution section, we include:

```
PLA1:-NEW pla(6,8,4,2,"TEST",8);
```

The above assignment of a NEW pla to the PLA reference variable cannot be inside a symbol definition. The drawing of the pla may be specified anywhere. To draw one of these PLAs at location (123,-208), we add:

```
draw(PLA1.planame,123,-208);
```

On the chip, we get something like this:



The '+' indicates the approximate origin for the PLA.

The following example shows a LAP program which includes a cell with two PLAs. The origin of the second PLA is placed 15 units to the left of the leftmost extremity of the first.

```
REF (pla) pla1,pla2;

! Note: these two statements cannot be placed inside ;
! a symbol definition. ;
```

```
pla1 :- NEW pla(2,2,2,1,"pla1",8);
pla2 :- NEW pla(2,2,2,1,"pla2",8);

define("cell1");

    draw(pla1.planame,0,8);
    draw(pla2.planame,pla1.mbb.ur QUA point.x + 15,2);

enddef;
```